(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
7 December 2000 (07.12.2000)

PCT

(10) International Publication Number
WO 00/74368 A2

(51) International Patent Classification[7]: H04N

(21) International Application Number: PCT/US00/15416

(22) International Filing Date: 1 June 2000 (01.06.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/324,073 1 June 1999 (01.06.1999) US

(71) Applicant: BSQUARE CORPORATION [US/US]; 3150 139th Avenue S.E., Suite 500, Bellevue, WA 98005-4081 (US).

(72) Inventors: BROOKS, Steve; 21433 SE 19th Street, Issaquah, WA 98029 (US). MURRAY, Jason; 23803 NE 27th Street, Redmond, WA 98053 (US). RICHARDS, David; 4155 145th Avenue, NE, Bellevue, WA 98007 (US).

(74) Agent: LARIVIERE, GRUBMAN & PAYNE, LLP; P.O. Box 3140, Monterey, CA 93942 (US).

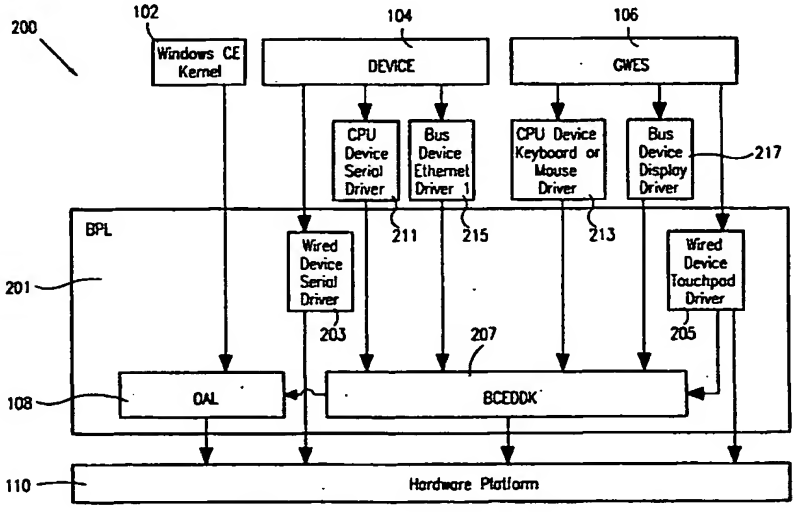
(81) Designated States (*national*): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

(84) Designated States (*regional*): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *Without international search report and to be republished upon receipt of that report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: DEVICE DRIVER PLATFORM LAYER



(57) Abstract: A Platform Layer interface for device drivers - also referred to hereinafter as the bSquaretm Platform Layer, or "BPL", - conforms a computers operating system - e.g., a WINDOWS CE operating system - and provides a transportable system for connecting device drivers to a variety of computer hardware platform. The BPL provides an interface that imposes structure on the operating system platform layer with several functionalities as individual components of the BPL used by the drivers, including: 1) a Memory Management component, 2) an Interrupt Allocation component, 3) an Interrupt Timers component, 4) a Direct Memory Access ("DMA") component, including slave and master bus operations, and 5) an Input-Output Access Routines component.

WO 00/74368 A2

DEVICE DRIVER PLATFORM LAYER

CROSS-REFERENCE TO RELATED APPLICATIONS

This patent application claims the benefit of U. S. Patent Application Serial No. 09/324,073, entitled "**DEVICE DRIVER PLATFORM LAYER**", filed June 1, 1999.

MICROFICHE APPENDIX

This patent application contains reference computer program listings and related material in the form of four (4) microfiche appendices A-D, containing a total of 143 frames, submitted in accordance with 37 C.F.R. 1.96; copies may be obtained by submission of a request to the U.S. Patent and Trademark Office with the appropriate fee as set forth in 37 C.F.R. 1.19.

BACKGROUND OF THE INVENTION

Field of the Invention.

The present invention relates generally to computer peripheral device driver controls, particularly to the Microsofttm WINDOWS CEtm operating system and device drivers compatible therewith and, more specifically, to a platform layer providing a universal interface in a WINDOWS CE operating system.

Description of Related Art.

It is known in the art that computers require a basic operating system - such as DOS (disk operating system), WINDOWS 98tm, WINDOWS NTtm, WINDOWS CEtm, or the like - to which other end user programs can be adapted in order for the user to load and use a variety of applications programs such as graphics programs. word processing programs, spreadsheet programs, and the like. The operating systems also use integrating programs to connect and

control a variety of hardware devices to the computer, such as printers, scanners, digital cameras, and the like. The program for integrating and performing computer peripheral device hardware control functions is commonly referred to as a "device driver." Device drivers can generally be categorized as graphics-oriented (e.g., for WINDOWS operating systems using the Microsoft provided graphics windowing event subsystem, "GWES"), such as video monitors, or all other peripheral devices, such as serial port devices, audio drivers, NDIS, and the like.

FIGURE 1 (Prior Art) is a graphical representation of the current WINDOWS CE operating system architecture 100 (more simply referred to as the "CE/OS" hereinafter). The Windows CE Kernel 102 program, provided by the Microsoft company houses a library of operating system function subroutines. Basically, it provides the standard hardware abstraction to support the WINDOWS CE Kernel 102, which dictates the required exported primitives. For the purposes of describing the present invention "exported primitives" shall mean basic functional units implemented by a software module and available for use by external software modules; and, "abstraction" shall mean a software representation of physical hardware or processes.

A Platform Layer 101 contains all source code and files required to run the CE/OS on a given Hardware Platform 110, typically, a sub-notebook computer, sometimes referred to as a "palmtop" computer. The Platform Layer 101 is the only hardware platform-dependent component of the CE/OS. The Platform Layer 101 includes loader technology for placing a CE/OS graphical user interface (GUI) image on the target Hardware Platform 110, an Original Equipment Manufacturer ("OEM") Adaptation Layer ("OAL") 108 for the CE Kernel 102, and specifically tailored device drivers 103, 211, 213, 215, 217 for any DEVICES 104 and GWES devices 106 to be connected to the Hardware Platform 110 - e.g., "Serial Port Driver 1," "Serial Port Driver 2," "Display Driver," "Touchpad Driver." OEM-OAL functions and structures provide a hardware abstraction that allows the CE/OS Kernel 102 to run only any platform. This is commercially available as the Microsoft Windows CE InfoViewer documentation provided by Microsoft. Except as specifically mentioned hereinafter, further description here is not necessary to an understanding of the present invention.

Peripheral DEVICE(s) 104 and GWES device(s) 106 are linked to the Hardware Platform 110 directly via respective device drivers 103, 211, 213, 215, 217. Therefore, each individual DEVICE 104 and GWES 106 requires a CE/OS compatible driver 103, 211, 213, 215, 217 for

each individual Hardware Platform 110.

Furthermore, in accordance with the CE Kernel 102, each device driver 103, 211, 213, 215, 217 must run as a Dynamic Link Library ("DLL") using industry standard inter-process control mechanisms - such as messages, call-backs, synchronization objects, shared memory. That means that the device driver 103, 211, 213, 215, 217 written by the OEM to the OAL 108 must include a Microsoft provided routine called the CE Device Driver Kit ("CEDDK"). It is still another rquirement of CE/OS that the Windows CE Kernel 102 must be linked with a library containing power up/down functions, interrupt functions, and serial port debug functions, and other common peripheral device functions as would be known in the art. It can now be recognized that the difficulty imposed is that every OEM - each having unique, proprietary features in their Hardware Platform 110, namely, different central processing units ("CPU") and firmware, or proprietary computer peripheral device 104, 106 to be used on such a Platform 110 - must both adapt to CE/OS and then still tailor every device driver 103, 211, 213, 215, 217 program individually. In other words, each target Hardware Platform 110 that runs CE/OS uses a unique Platform Layer 101. As target Hardware Platforms that execute CE/OS get larger and more complex, the amount of time required to design and test an OAL 108 increases significantly.

There is a need for simplification of device driver development for WINDOWS CE operating system interface. It has been found that by providing certain universal functions to the OEM, there is a significantly decrease in the amount of time needed to adapt a new Hardware Platform and computer peripherals to CE/OS.

SUMMARY OF THE INVENTION

In its basic aspects, the present invention provides a method for facilitating communication between computer platforms, each of the computer platforms using a like predetermined operating system, and computer peripheral devices, each of the devices having a respective individual device driver system, via a platform layer interface system, the platform layer interface system including a device driver adaptation subsystem associated with the predetermined operating system. The method includes the steps of: providing a device driver adaptation subsystem having a library of computer hardware functional abstractions wherein the

functional abstractions are substantially common to a plurality of differing types of the computer platforms; providing a modified platform layer interface system having the device driver adaptation subsystem interfaced with the device driver adaptation subsystem; and interfacing communications between respective the hardware platforms and the computer peripheral devices, respectively, via the modified platform layer interface system using the device driver adaptation subsystem such that each the device driver system written to the device driver adaptation subsystem is transportable across a plurality of differing types of the computer platforms.

In another basic aspect, the present invention provides a computerized apparatus for conducting interfaced operations between a computer platform, having a predetermined computer operating system wherein the operating system has a computerized platform layer interface system including adaptation mechanisms for conforming computer peripheral devices to operate in conjunction with the computer platform, and computer peripheral devices, each having a device driver system. The computerized apparatus includes: mechanisms for providing a device driver adaptation subsystem having a library of computer hardware functional abstractions wherein the functional abstractions are substantially common to a plurality of differing types of each the computer platform; mechanisms for providing a modified platform layer interface system having the device driver adaptation subsystem therein interfaced with the device driver adaptation subsystem; and mechanisms for interfacing communications between the computer platform and the computer peripheral devices connected thereto, respectively, via the modified platform layer interface system using the device driver adaptation subsystem such that each the device driver system written to the device driver adaptation subsystem is transportable across the plurality of differing types of the computer platforms.

In another basic aspect, the present invention provides a computer-to-peripheral platform layer construct for interfacing peripheral device drivers to a variety of types of computer hardware platforms, each of the platforms having a common operating system, including: associated with the operating system, predetermined adaptation layer mechanisms for adapting a peripheral device to a predetermined one of the computer platforms; and interface mechanisms for providing an interface between the drivers and each of the types of computer hardware platforms, the interface mechanisms, including memory management mechanisms for allocating and for mapping memory substantially concurrently to a call for memory by each of the drivers, interrupt mechanisms for handling system interrupt allocation and for connecting each of the

driver to any free interrupt vector in the operating system substantially concurrently to a system interrupt for each of the drivers, direct memory access mechanisms for abstracting platform direct memory access requirements for adaptation by each of the drivers, and input-output mechanisms for interfacing the platform layer construct and each of the platforms, wherein the construct provides a transparent interface having localized computer hardware functionality such that any of the device drivers programmed to the construct is transportable to a plurality of hardware platforms.

It is an advantage of the present invention that it provides an universal device driver interface for CE/OS. It is an advantage of the present invention that it provides a centralized management of system resources across devices drivers and the OAL 108, reducing the possibility of desynchronization between a peripheral device and the CE/OS. It is another advantage of the present invention that is decreases the amount of time needed to adapt CE/OS to a Hardware Platform 110. It is a further advantage of the present invention that a device driver needs to be compiled only once per instruction set architecture rather than once per Hardware Platform 110. It is still another advantage of the present invention that its use in the development of a device driver results in a portable driver; that is, porting CE/OS adaptations to new Hardware Platforms is facilitated. It is yet another advantage of the present invention that it can be distributed in a computerized apparatus, such as a memory disk or CD-ROM, in binary form, allowing the loading of device drivers in an already deployed operating system. It is an advantage of the present invention that it abstracts common operations, such as memory mapping and allocation, interrupt allocation, interrupt timing, and direct memory access, reducing the amount of new and redundant code required for each new compatible device driver. It is another advantage of the present invention that it includes functions for drivers to manage contiguous blocks of physical memory, for example, to accommodate the frame buffer of a video driver.

Still another advantage of the present invention is that bus device drivers are provided with a standard set of hardware abstractions, permitting use with any CPU architecture, CPU implementation, or target Hardware Platform 110 having the same bus architecture. A further advantage of the present invention is that it allows all platform layer implementation code to be structured similarly, making installation and documentation easier.

The foregoing Summary and list of advantages is provided for the convenience of the reader; the inventor does not intend that it be exclusive of other implementations or

characteristics nor that it be construed as a limitation on the scope of the invention and no such intention should be implied.

Other objects, features and advantages of the present invention will become apparent to persons skilled in the art upon consideration of the following explanation and the accompanying drawings, in which like reference designations represent like features throughout the drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIGURE 1 (Prior Art) is a graphical depiction of a generic Hardware Platform adapted for use with the Microsoft WINDOWS CE Operating System.

FIGURE 2 is a graphical depiction of the present invention as conformed to the WINDOWS CE Operating System structure.

FIGURE 3 is a graphical depiction of the BCEDDK component of the present invention as shown in FIGURE 2.

FIGURE 4 is a flow chart illustrating a physical memory allocation subroutine in accordance with the present invention as shown in FIGURE 3.

FIGURE 5 is a flow chart illustrating an interrupt processing subroutine in accordance with the present invention as shown in FIGURE 3.

FIGURE 6 is a flow chart illustrating a slave direct memory access, without auto-initialization, subroutine in accordance with the present invention as shown in FIGURE 3.

FIGURE 7 is a flow chart illustrating a slave direct memory access, with auto-initialization, subroutine in accordance with the present invention as shown in FIGURE 3.

FIGURE 8 is a flow chart illustrating a BusMaster direct memory access, with auto-initialization, subroutine in accordance with the present invention as shown in FIGURE 3.

The drawings referred to in this specification should be understood as not being drawn to scale except if specifically noted.

DESCRIPTION OF THE PREFERRED EMBODIMENT

Reference is made now in detail to a specific embodiment of the present invention, which illustrates the best mode presently contemplated by the inventors for practicing the invention. Alternative embodiments are also briefly described as applicable. Subtitles are used hereinafter for mere convenience and are not intended as any limitation on the scope of the invention nor

should any such intention be implied. In an exemplary embodiment, the present invention is described as employed with a WINDOWS CE operating system; use of this exemplary embodiment is not intended by the inventors to be a limitation on the scope of the invention nor should any such limitation be implied therefrom.

bSquare Platform Layer 207 - General

FIGURE 2 is a graphical depiction of the present invention CE/OS device driver system 200, providing a universal interface - also referred to hereinafter as the bSquaretm Platform Layer, or "BPL," 201 - conformed to the WINDOWS CE Operating System Structure for a generic Hardware Platform 110. A microfiche appendix to this Detailed Description provides programming details in terms of object code, syntax, parameters, returns, and the like.

The existing Windows CE Kernel 102, a peripheral DEVICE 104, and GWES peripheral 106 are given to be the same exemplary apparatus as shown in the prior art of FIGURE 1. The BPL 201 provides an interface that imposes structure on the CE/OS platform layer. The BPL 201 incorporates the OAL 108, as in the prior art of FIGURE 1, as a static library. The BPL 201 further includes a restructured CEDDK, referred to also as the bSquare-CEDDK, or more simply the "BCEDDK," 207 component. For purposes of this document, "component" is defined as referring to any library (dynamic or static) that performs well-defined operations and exports a well-defined applications program interface ("API") set.

The BPL architecture divides CE/OS devices 104, 106 as being in one of several basic functional categories:

1) CPU Device Drivers 211, 213 are drivers for devices that exist as part of the CPU chip set core or are integrated on the same silicon chip as the CPU - that is, CPU Devices are CPU-dependent and exist only when the CPU is installed on the target Hardware Platform 110, e.g., a video controller included on a CPU chip;

2) Bus Device Drivers 215, 217 are drivers for peripheral devices that reside on an architected bus (e.g., ISA, PCI and USB), but are CPU independent; and, 3) Wired Device Drivers 203, 205 are peripheral devices connected to the CPU over a non-standard bus and are intimately tied to the Platform 110.

Comparing FIGURE 2 with FIGURE 1, note that in accordance with the present invention, common drivers such as "Serial Port Driver 1 and Driver 2" of FIGURE 1 which

communicated directly with the Hardware Platform 110 now have an indirect interface through API's of the present invention provided by the BCEDDK 207, namely shown as "CPU Device Serial Driver" 211 and "CPU Device Keyboard or Mouse Driver" 213 of FIGURE 2. Hardware Platforms that share the same CPU share the same CPU Device Drives. Therefore, any core CPU driver code developed with BPL is usable on all Hardware Platforms using that CPU.

Similarly, Bus Device Drivers, e.g. "Bus Device Ethernet Driver 1" and "Bus Device Display Driver," now have an indirect interface through the API's provided by the BCEDDK 207. By providing the BCEDDK 207 component, once a particular Bus Device Driver is written with the BPL 201, it may be shared by simply recompiling the driver for a different instruction set architecture.

Because each individual "wired device," such as a touchpad having a chip wired to the general-purpose input-output ("I/O") pins on the CPU, is a highly proprietary device, no assumptions can be made about connections or communication protocols. Thus, no standardized interface can be developed and wired devices still communicate directly with the Hardware Platform 110, shown in FIGURE 2 as "Wired Device Serial Driver" 203 and "Wired Device Touchpad Driver" 205. However, accommodations allow operation within the BPL 201; that is, for wired device drivers, coordination with the BCEDDK 207 component of any interrupts, physical memory locations, or DMA channels which the wired drivers might use in order to prevent incorrect sharing of resources is provided.

BCEDDK

FIGURE 3 is a graphical depiction of the components as integrated into the BCEDDK 207. Subroutines corresponding to further information found in the microfiche appendix are designated by being enclosed in brackets, [ ] , hereinafter.

The BCEDDK 207 component provides localization of functionality such that OEMS of a Hardware Platform 110 or a peripheral device 104, 106 need program only to the BCEDDK.

That is, the BCEDDK 207 component provides a library of application program interfaces ("API") for fully abstracting bus and CPU device drivers. In other words, a Device Driver 301 written to BCEDDK 207 is transportable across different CE/OS hardware platforms that are also BCEDDK compatible. To accomplish this, the BCEDDK 207 provides several functionalities as individual components of the BPL 201 used by the CE/OS Device Driver 301:

1) a Memory Management component 303 (see also microfiche Design Guide Appendix

A., Memory Manager Functions, for syntax, fields, parameters, and the like),

2) an Interrupt Allocation component 305 (see also Microfiche Design Guide Appendix B., Interrupt Processing Functions),

3) an Interrupt Timers component 307 (see also Microfiche Design Guide Appendix B., Interrupt Processing Functions),

4) a Direct Memory Access("DMA") Abstraction component 309, including slave and master bus operations (see Microfiche Design Guide Appendix C.), and

5) an Input-Output Access Routines Abstraction component 311, adapted directly from CE/OS (see also Microfiche Design Guide Appendix C).

The microfiche Design Guide Appendix D provides data structures used by the BPL, specifically the attributes of a specific device 104, 106 corresponding to a retrievable adapter object, as defined hereinafter.

Memory Management 303 component

Creating specific mappings between physical and virtual memory is one of the most common memory operations performed by device drivers; memory mapped registers and memory buffers in physical memory must be mapped into a virtual address space to be accessed. CE/OS device drivers run as a DLL in a user space process where memory accesses map into the virtual address space. The Hardware Platform 110 (generally with its own CPU memory management unit ("MMU")) and the CE/OS must map between virtual address space and physical memory.

In general, the BCEDDK Memory Management 303 component provides universal functions necessary for allocating and mapping memory. Memory Management 303 centralizes a particular Device Driver 301 requirements for random access memory.

Without this function, namely under a strict CE/OS only scheme, a block of memory in the Hardware Platform 110 would always have to be reserved for device drivers.

Turning now to FIGURE 4, a device driver initialization, step 401, provided by the device OEM sets up the device 104, 106 and its Device Driver 301 program for interfacing with the system. Once a device 104, 106 is initialized, its Driver 301 sends a request for a block of physical memory to BPL, step 403. The CE/OS does not provide functions for handling contiguous blocks of physical memory. A subroutine, [MmAllocatePhysicalMemory] obtains

and allocates a range of physically contiguous, cache-aligned memory from a non-paged memory pool of the Hardware Platform 110; once obtained, this function sends the Driver 301 a mapped, virtual, address pointer to the base virtual address for the allocated memory to the Device Driver 301 and writes a pointer to a physically contiguous block of memory of the requested size, step 405.

This block of memory is positioned below the maximum acceptable physical address and conforms to the requested alignment requirement. Because the virtual pointer returned is associated directly with a physical address range, the pages comprising the buffer are locked. If no memory is currently available, a NULL flag is set and the Device Driver 301 notified. A [MmFreePhysicalMemory] subroutine releases a range of previously allocated memory.

Providing further memory management support, the BCEDDK 207 furnishes function for handling memory descriptor lists. A memory descriptor list ("Mdl") is a data structure that completely describes a contiguous virtual buffer in terms of the physical pages that comprise the buffer. The Mdl keeps track of the virtual base of the buffer, the buffer's size, and the offset into the first physical page where the buffer begins. A Device Driver 301 uses Mdl's when it needs to know the physical pages that make up a virtual buffer. The Mdls dictate virtual-to-physical and physical-to-virtual address tracking. A [MmCreateMdl] subroutine allocates a new memory descriptor list, describing either a virtual, contiguous user buffer or a common buffer, by initializing an information data header, locking the corresponding buffer, and filling in the corresponding physical pages for each virtual page in the described buffer. A pointer is provided to the Device Driver 301 of the initialized Mdl. The physical pages in the buffer described by the Mdl are locked. A [MmFreeMdl] subroutine merely frees a previously allocated Mdl. Once the Mdl is created, the Device Driver 301 can use other functions. A [MmGetMdlByteCount] retrieves and informs the Device Driver 301 of the length of the buffer described by the Mdl. A [MmGetMdlByteOffset] subroutine retrieves any offset in the page of the buffer described by the Mdl, returning a byte offset to the Device Driver 301. A [MmGetMdlStartVa] subroutine retrieves starting virtual address - the virtual address of the buffer that the Mdl describes rounded to the nearest page - of the Mdl and provides a pointer to that starting virtual address to the Device Driver 301. A [MmGetMdlVirtualAddress] subroutine retrieves the virtual address of the buffer described by the Mdl and returns a pointer to that virtual address to the Device Driver 301.

In addition to a memory descriptor list, the Device Driver 301 requires information for mapping from virtual memory to physical memory. That is, memory mapped registers and memory buffers in physical memory must be mapped to a process' virtual address space to be accessed. A [MmMapIoSpace] subroutine, initiated by the Device Driver 301 call for mapping, specifies the starting physical address and size of the I/O range to map and whether the memory is a cache. A virtual pointer to the base virtual address that maps to the base physical addresses for the specified range is returned to the Device Driver 301; as this pointer is associated directly with a physical address range, the pages comprising the buffer are locked. If space for mapping the range is insufficient, a NULL signal is returned in place of the pointer. A [MmUnmapIoSpace] releases a specified range of physical addresses mapped by [MmMapIoSpace].

Interrupt Processing 305 component

In the CE/OS, each system interrupt signal ("SYSINTR") is assigned to a certain bus interrupt. Windows CE/OS limits the number of SYSINTRs to twenty-four; that is, in the standard implementation of CE/OS compatible applications it is not possible to provide drivers with access to more than twenty-four interrupts on one Hardware Platform 110.

In accordance with the present invention, as illustrated by FIGURE 5, while the system is still limited to twenty-four SYSINTRs, BPL allows Device Drivers 301 to access to an arbitrary number of interrupts, allowing a Device Driver 301 to allocate and connect to any free interrupt vector in the OS. A driver is only interested in vectors for the particular architected bus on which it runs. BCEDDK provides an Interrupt Allocation 305 abstraction to relate the bus to SYSINTR's.

An [InterruptConnect] subroutine allocates a SYSINTR and associates it with a specified Hardware Platform 110 system interrupt vector; it returns a valid SYSINTR value to the Device Driver 301 that is unique in terms of a bus-related interrupt vector. An [InterruptDisconnect] subroutine disconnects the specified SYSINTR and disassociates it from the vector specified in the call to the [InterruptConnect] subroutine when the driver has finished performing interrupt processing.

The process can be generalized as follows. Following driver initialization, step 401, the Device Driver 301 requests a SYSINTR by specifying which bus interrupts the Driver wants to

be notified of, step 501. BCEDDK 207 updates a table that associates a SYSINTR with a particular bus interrupt, step 503.

The Device Driver 301 associates the SYSINTR with an event and enters a wait state, step 505. BCEDDK 207 is called whenever a CPU's interrupt occurs, step 507. BCEDDK 207 determines which bus interrupt occurred and what SYSINTR that interrupt is associated with, reporting to the Driver if it is the associated SYSINTR, step 509. The Device Driver 301 releases the wait state and processes the SYSINTR, step 511. Once processed, step 513, the Device Driver 301 returns to the wait state for the next associated SYSINTR. Thus, BCEDDK 207 specifies an interrupt in terms of a bus-related interrupt vector rather than in terms of a system interrupt vector.

Interrupt Timers 307 component

Standard OAL 108 timers for Interrupt, Disable, Done, Enable, and Initialize are adapted for BPL 201. Additional TIMERS 307 are provided for specific BCEDDK 207 functions. To provide an interface for the hardware timers, the BCEDDK 207 defines both a timer object and a set of routines to manipulate that object. The timer object consists of four main parts.

1) Hardware timer: Only one timer object may be allocated for each hardware timer, which is specific to the CPU and Platform 110. Each timer has a given granularity, e.g., of 100 ns, subject to CPU and Platform 110 limitations. Behavior of timer objects during suspended power state is Platform 110 dependent, as hardware timer behavior depends on the Platform's power state.

2) Up counter: Represents the number of given timer object intervals, e.g., 100 ns, the timer runs. When the timer object is stopped, the up counter is frozen until the timer object is restarted. In the preferred embodiment, the up-counter is a 64-bit unsigned integer.

3) Period: Specifies at what frequency a timer will generate interrupts. This value must be set to ensure that the system is not overloaded with interrupts. In the preferred embodiment, the maximum period is 232 * 100 ns (429.5 seconds).

4) SYSINTR: The interrupt value allocated when the timer object has been running continuously for multiple periods.

If the timer object is not allocated, the SYSINTR value is irrelevant. The timer object has three states: running, stopped, and unallocated. If unallocated, the timer object does nothing. If

allocated, the timer object is either running or stopped.

When running, its up counter adds up each elapsed timer object interval, generating interrupts on the specified SYSINTR at the specified period. When stopped, it retains its current value but no longer generates interrupts.

An [InterruptConnectTimer] subroutine is provided for allocating a new SYSINTR for a system timer. It is used to connect to timer interrupt sources. An [InterruptDisconnectTimer] subroutine disassociates the specified SYSINTR from its corresponding timer. The SYSINTR and timer can then be reallocated. An [InterruptStartTimer] subroutine starts the timer associated with the specified SYSINTR. This routine is only valid to call on a SYSINTR allocated by the [InterruptConnectTimer]. An [InterruptStopTimer] subroutine stops the timer associated with the specified SYSINTR. The timer stops generating interrupts but all resources remain allocated. An [InterruptQueryTimer] subroutine reads the timer associated with the specified SYSINTR and returns how many intervals the timer has run (e.g., a given number of 100 ns intervals). [InterruptConnectTimer] and [InterruptDisconnectTimer] subroutines operate much like [InterruptConnect] and [InterruptDisconnect], the difference being that the [InterruptConnectTimer] subroutine does not need a specified interrupt vector in order to connect, but will allocate from a pool of timer objects. Once the timer object has been allocated, and a SYSINTR corresponding to that object returned, the SYSINTR must be associated with an event by calling [InterruptInitialize]. Only after a timer object has been allocated and its SYSINTR associated with an event is it valid to call the other timer routines. Following is an example that sets up a periodic interrupt (e.g., of 5 ms) and frees the timer object via the present invention.

EXAMPLE 1:

1. The Device Driver 301 calls [InterruptConnectTimer] to allocate a timer object and a SYSINTR.

2. The Device Driver 301 calls [InterruptInitialize] with the returned SYSINTR and an event at which it is to be set when the SYSINTR occurs.

3. When the exemplary 5 ms periodic interrupts are to start, the Device Driver 301 calls [InterruptStartTimer] with the SYSINTR and a period of 50000. The timer object will begin generating interrupts every 5 ms.

4. When an interrupt occurs, the timer automatically resets and will generate another

interrupt in 5 ms regardless of any processing by the IST that is handling the event associated with the SYSINTR.

5. When the periodic interrupts are to stop, the Device Driver 301 calls [InterruptStopTimer] with the SYSINTR.

6. When the Driver 301 is finished with the timer object (e.g., is being unloaded) [DisableInterrupt] should be called with the SYSINTR. This disables the timer interrupts. The timer routines are no longer valid from this point on.

7. Finally, the Device Driver 301 should call [InterruptDisconnectTimer] to free the timer object.

Memory Hardware Abstraction 309 component

Devices 104 that require large data throughput in short periods of time cannot rely on the Hardware Platform 110 CPU to move the data as it would degrade overall system performance. Direct Memory Access is therefore employed. Generally it is known to use known manner Packet and Common Buffer BusMaster DMA or Packet and Common Buffer Slave DMA, where slave modes use the Hardware Platform 110 system, or an auxiliary processor, DMA controller and BusMaster modes arbitrate for a system bus by having device 104, 106 proprietary DMA hardware to move data between Platform 110 memory and the device 104, 106. DMA code is generally complex. The CE/OS does not provide any mechanism for allocating DMA channels and setting up DMA operations for slave mode DMA. A Device Driver 301 that needs to use a slave mode DMA must make assumptions about the Hardware Platform 110.

BCEDDK 207 removes Hardware Platform 110 dependencies from the DMA code, providing a universal DMA Abstraction 309 by platform-dependent representations that are transparent to the Device Driver 301 in the form of adapter objects, map registers, and adapter channels. Adapter objects are data structures that describe the available hardware that the Device Driver 301 needs to perform DMA operations. The information that BCEDDK 207 needs to allocate or set up hardware is contained in the adapter object. A map register represents a mapping from a bus-related address to a system-accessible physical address. [The actual function that a map register performs is hardware platform-dependent; some platforms do not use map registers while others use hardware register to map bus addresses to system-accessible physical addresses and others maintain a virtual map.] To actually perform a DMA operation via

BCEDDK 207, a Device Driver 301 must allocate an adapter channel or a common buffer, where an adapter channel allocation represents ownership of a system DMA controller channel and the availability of map registers. Such allocation prevents multiple drivers from trying to use the same DMA controller at the same time.

Slave Mode DMA Modes

For a slave mode DMA operation, BCEDDK 207 maintains the complete state of the system DMA controller, divorcing the Device Driver 301 from controller accesses and making the driver platform independent.

Packet-based slave mode DMA does not use the controller standard Auto-Initialize mode. The DMA controller is set up to transfer on each page in the buffer being sent or received. The packet-based slave mode DMA process can be generalized as follows and as shown in FIGURE 6. The Device Driver 301 initialization routine, step 401, is called with the Driver informing BCEDDK 207 of the packet-based slave mode DMA choice and which DMA channel the Driver will use, step 601; this handshake also negotiates the maximum DMA transfer size during one DMA operation, step 603. Following the handshake, the Device Driver 301 signals readiness to transfer data packets to or from a buffer, step 605. If the DMA channel is not available, step 607-NO, a WAIT state is entered, step 609. When no other Driver is using the DMA channel, step 607-YES, BCEDDK 207 acknowledges access availability. The Driver calculates whether all or some of the data can be transferred based upon the handshake negotiation. When ready, the Device Driver 301 instructs the BCEDDK 207 to begin the DMA transaction on the slave DMA controller, step 611. If the Driver's buffer is outside of a memory area that the DMA controller can reach, the BCEDDK 207 copies the buffered data to accessible memory. The BCEDDK 207 then programs the DMA controller to make the transfer, step 613. When complete, step an interrupt is sent to the Device Driver 301. Depending on the size of the data transfer, the next transfer is initiated, step 615-NO, or, step 615-YES, the Driver waits, step 617, for its next DMA operation. The common buffer slave DMA routine uses the system DMA controller with the Auto- Initialize mode enabled as depicted in FIGURE 7. The Driver sets up the DMA controller to continually read from a common buffer that is accessible from both the CPU of the Hardware Platform 110 and the Device Driver 301.

The common buffer slave mode DMA operation can be generalized as follows. The

Device Driver 301 initialization routine, step 401, is performed. The Driver informs BCEDDK 207 of the slave mode DMA with Auto- Initialize choice, step 701. The Drive call includes which DMA channel the Driver will use; this handshake also negotiates the maximum DMA transfer size during one DMA operation, step 703, and Device Driver 301 requests allocation of a buffer accessible from the Driver and the bus. When ready to transfer data to or from the allocated buffer, the Device Driver 301 signals readiness to BCEDDK 207, step 705. If the channel is occupied, step 707-NO, a WAIT state is initiated, step 709. When no other driver is using the DMA channel, step 707-YES, BCEDDK 207 authorizes exclusive access to the Driver. The Device Driver 301 signals the BCEDDK 207 to begin the DMA transmission on using the slave DMA controller, step 711. Again, the BCEDDK 207 programs the slave controller, step 713. The transmission initiates and continues, step 715, until the end of the buffer in which case the DMA controller will reset to the beginning of the buffer; the Device Driver 301, having exclusive access at this time, is free to copy data into or out of the buffer as need. Once finished, the Driver 301 instructs BCEDDK 207 to stop DMA control and WAITS for the next DMA operation, step 717. The following is an example of the Device Driver 301 performing a packet-based Slave DMA via the present invention.

EXAMPLE 2:

1. The Device Driver 301 calls [HalGetAdapter] to allocate an adapter object. The driver specifies a DEVICE_DESCRIPTION structure. For a packet-based slave DMA device, the driver must set Master to FALSE and Auto-Initialize to FALSE.

2. The Device Driver 301 calls [MmCreateMdl] to create an Mdl. The function locks down the virtual buffer and determines the physical pages that comprise the buffer to access. When the actual DMA transfer is about to commence, the BCEDDK 207 uses these physical age numbers to set up the hardware.

3. The Device Driver 301 calls [HalAllocateAdapterChannel] when ready to set up the device 104, 106 for DMA transfer. The device 104, 106 must have exclusive access to needed map registers and the DMA controller. When the driver requests these from [HalAllocateAdapterChannel], the function blocks until the resources are available.

4. The Device Driver 301 calls [HalMapTransfer] to set up the system DMA hardware to perform the DMA.

5. Step 4 is repeated until the entire buffer has been transferred.

6. To release the adapter channel for other drivers to use, the Device Driver 301 calls [HalFreeAdapterChannel].

7. The Device Driver 301 calls [MmFreeMdl] when finished with the MDL.

The transfer operation for packet-based Slave DMA operation in accordance with the present invention is exemplified as follows.

EXAMPLE 3:

1. The Device Driver 301 calls [HalMapTransfer] to instruct the hardware to perform a DMA operation. Inputs to the function include the adapter object, the map registers, the virtual buffer MDL, and an index specifying how much of the buffer has been transferred.

2. When [HalMapTransfer] completes, it updates the virtual index value by the amount of data transferred. HalMapTransfer does not use this pointer to access data. It is used as an index into the MDL.

3. In its first call to [HalMapTransfer], the Device Driver 301 should provide a virtual pointer to the head of the buffer being transferred. After calling this function, the Driver 301 must call [HalFlushAdapterBuffers] to ensure that any cached data in the DMA controller has been flushed. Slave DMA devices ignore the address returned from [HalMapTransfer].

4. The Device Driver 301 now repeats the calls to [HalMapTransfer] and [HalFlushAdapterBuffers] as needed until the entire buffer is transferred. On each of the repeat calls to [HalMapTransfer], the Driver 301 should provide the virtual index value returned from the preceding call to [HalMapTransfer].

The following is an example of the Driver 301 performing a common buffer Slave DMA call via the present invention.

EXAMPLE 4:

1. Upon loading, the Device Driver 301 calls [HalGetAdapter] to allocate an adapter object. The driver specifies a DEVICE_DESCRIPTION structure. For a common buffer slave DMA device, the Driver 301 must set Master to FALSE and Auto-Initialize to TRUE.

Also at startup, the Device Driver 301 calls [HalAllocateCommonBuffer] to allocate a buffer common to both the CPU and the device.

2. If [HalAllocateCommonBuffer] returns NULL, the Device Driver 301 should free resources, unload, and report failure.

3. The Device Driver 301 calls [MmCreateMdl] to create an MDL for the common buffer. The function locks down the virtual buffer and determines the physical pages that comprise the buffer to access. When the actual DMA transfer is about to commence, the BCEDDK 207 uses these physical page numbers to set up the hardware.

4. If DMA is from the CPU to the device, data is moved into the common buffer created by [HalAllocateCommonBuffer] (see step 2).

5. The Device Driver 301 calls [HalAllocateAdapterChannel] when ready to set up the device for DMA transfer. The device 104, 106 must have exclusive access to needed map registers and the DMA controller, which the Driver 301 requests from [HalAllocateAdapterChannel]. The function blocks until these resources are available.

6. The Device Driver 301 calls [HalMapTransfer] to set up the system DMA hardware to perform the DMA.

7. The Device Driver 301 calls [HalFreeAdapterChannel] to release the adapter channel.

8. The Device Driver 301 calls [MmFreeMdl] when finished with the MDL. The following is an example of a transfer operation for a common buffer Slave DMA operation in accordance with the present invention.

EXAMPLE 5:

1. The Device Driver 301 calls [HalMapTransfer] one time to instruct the hardware to perform a DMA operation. Slave DMA devices ignore the address returned from [HalMapTransfer].

2. After [HalMapTransfer] returns, the Driver 301 should call [HalReadDmaCounter] to find out how much data still needs to be transferred. If required, the driver should copy data to or from the common buffer.

3. When the DMA transfer is complete, the driver must call [HalFlushAdapterBuffers] to ensure that any cached data in the DMA controller has been flushed. BusMaster DMA Mode CE/OS does not provide any mechanism for setting up BusMaster DMA data transfers. A Device Driver 301 that need to use BusMaster DMA operations must make assumptions about the Hardware Platform 110 it is running on.

For a BusMaster DMA device, since the device 104, 106 has a proprietary DMA controller and the Device Driver 301 has the proprietary code for the proprietary DMA controller, and thus is already platform-independent, BCEDDK 207 merely obtains mappings from bus addresses to system memory. The Device Driver 301 sets up its DMA controller for a transfer of each page of the data being sent or received.

The following types of subroutines are used in the DMA Abstraction 309, the details of which are set forth in Microfiche Design Guide Appendix C: [HalAllocateAdapterChannel]: allocates an adapter channel for DMA; this function blocks until the requested adapter channel is available; the driver uses the value returned - viz., TRUE if the adapter is allocated and FALSE otherwise - to call the [HalMapTransfer] subroutine; [HalAlloctaeCommonBuffer]: allocates memory and maps it to be simultaneously accessible from both the Platform 110 and the Device; the return is a pointer to the virtual address of the allocated range of memory or a NULL if a buffer cannot be allocated; [HalFlushAdapterBuffer]: flushes the buffer for the specified adapter; the return is a TRUE if the buffer is transferred to the Device, FALSE otherwise; [HalFreeAdapterChannel]: releases a DMA controller for a Device when the Driver has completed its DMA operation; [HalFreeCommonBuffer]: releases a common buffer; [HalFreeMapRegisters]: releases map register resources; [HalGetAdapter]: retrieves an adapter object corresponding to the described Device; the return is a pointer to the requested adapter object representing the BusMaster adapter or DMA controller channel, or a NULL if it cannot retrieve the adapter object; [HalGetBusDataByOffset]: retrieves details about a specified memory slot or address and returns the number of bytes of data in the specified buffer; [HalGetDmaAlignmentRequirement]: retrieves the size of a cache memory boundary and returns the information to the Driver; [HalMapTransfer]: creates the map for a DMA transfer as a locked buffer described by the specified Mdl; this function sets up a number of map registers (up to the maximum retrieved by [HalGetAdapter]) for the given adapter object, and returns the logical address of the region mapped for a BusMaster adapter (Drivers that use system DMA controller adapters ignore this address); [HalReadDmaCounter]: retrieves the number of bytes remaining to be transferred in a DMA transfer for a buffer, returning the number; [HalTranslateBusAddress]: translates a bus-specific address into a system logical address, returning TRUE if the translation is successful and FALSE otherwise. The BusMaster DMA process can be generalized as follows and as illustrated in FIGURE 8. The Device Driver 301

initializes, step 401. The Device Driver 301 informs the BCEDDK 207 that it is about to perform a BusMaster DMA operation, step 801; again, this handshake also negotiates the maximum DMA transfer size during one DMA operation.

The Driver can choose whether to allocate its own buffer, step 803-YES, or have BCEDDK 207 allocate a physical buffer, steps 803-NO and 805 (see FIGURE 4). When the Device Driver 301 is ready to begin a DMA transaction, it requests BCEDDK 207 take control of the selected buffer, step 807 (providing the address, insuring that the buffer used in the DMA transaction is smaller than the maximum DMA transfer size negotiated if allocating its own buffer). BCEDDK 207 establishes mapping registers or calculates Hardware Platform 110 specific offsets and reports where the buffer is addressable on the Device's bus, step 809.

If the Device Driver 301 only needs to use one DMA buffer, then the DMA transaction is performed, steps 811-YES and 813; if several buffers are required, step 811-NO, the request and mapping steps are repeated (for packet-based BusMaster DMA, if the Device has scatter/gather support, the driver sets up its DMA controller to transfer multiple pages; if not, the Driver transfers one page at a time), before transfer, step 813. One the DMA is completed, a WAIT state, step 815, is entered until the next BusMaster DMA call, step 801.

Note that a Device Driver 301 must create an adapter object for each slave DMA controller and for each BusMaster device in the operating system.

The following is an example of a typical calling sequence for packet-based BusMaster DMA to or from a virtual buffer.

EXAMPLE 6:

1. Upon loading, the Device Driver 301 calls [HalGetAdapter] to allocate an adapter object. The Driver 301 specifies a DEVICE_DESCRIPTION structure. For a packet-based BusMaster DMA device, the driver must set Master to TRUE.

2. To start DMA to or from a virtual buffer, the Device Driver 301 calls [MmCreateMdl] to create an MDL. The function locks down the virtual buffer and determines the physical pages that comprise the buffer to access. When the actual DMA transfer is about to commence, the BCEDDK 207 uses these physical page numbers to set up the hardware.

3. The Device Driver 301 calls [HalAllocateAdapterChannel] when ready to set up the device for DMA transfer. The device 104 must have exclusive access to needed map registers, which the Driver 301 requests from [HalAllocateAdapterChannel]. The function blocks until the

registers are available.

4. The Device Driver 301 calls [HalMapTransfer] to set up map registers for the DMA transfer.

5. Step 4 is repeated until the entire buffer has been transferred.

6. The Device Driver 301 calls [HalFreeMapRegisters] to release the map registers allocated through [HalAllocateAdapterChannel].

7. The Device Driver 301 calls [MmFreeMdl] when finished with the MDL.

The following is an example of a transfer operation in accordance with the present invention for packet-based BusMaster DMA.

EXAMPLE 7:

1. The driver the Device Driver 301 calls [HalMapTransfer] with a value of NULL for an adapter object. In BusMaster devices, the DMA hardware is in the device 104 itself so [HalMapTransfer] only sets up the map registers. [HalMapTransfer] returns the bus address and the length of the region mapped. The Driver 301 can use this information to setup its DMA hardware.

2. If the device 104 has scatter/gather support, the Driver 301 calls [HalMapTransfer] multiple times. In its first call, the Driver 301 should provide a virtual pointer to the head of the buffer being transferred. After calling this function, the Driver 301 must call [HalFlushAdapterBuffers] to ensure that any cached data has been flushed. The Device Driver 301 now repeats the calls to [HalMapTransfer] and [HalFlushAdapterBuffers] as needed until the available map registers have been exhausted or until the scatter/gather hardware is full.

Once the scatter/gather registers have been loaded, the Driver 301 should instruct the device 104 to start its DMA. On each of the repeat calls to [HalMapTransfer], the Driver 301 should provide the virtual index value returned from the preceding call to [HalMapTransfer].

3. If the device 104 does not have scatter/gather support, the Driver 301 calls [HalMapTransfer] only once. Then the Driver 301 should instruct the Device 104 to begin DMA, based on the bus address returned from [HalMapTransfer]. When DMA transfer is complete, the driver must call [HalFlushAdapterBuffers] to ensure that any cached data in has been flushed.

The following is an example of the Driver performing a common buffer BusMaster DMA:

EXAMPLE 8:

1. The Device Driver 301 calls [HalGetAdapter] to allocate an adapter object. The Driver 301 specifies a DEVICE_DESCRIPTION structure (appendix D). For a common buffer BusMaster DMA device, the Driver 301 must set Master to TRUE. Also at initialization, the Driver 301 calls [HalAllocateCommonBuffer] to allocate a buffer common to both the CPU and the device. The virtual pointer returned from [HalAllocateCommonBuffer] can be used by the Driver 301 to move data directly to or from the buffer with the CPU. If [HalAllocateCommonBuffer] returns NULL, the Driver 301 should free resources, unload, and report failure.

2. After a common buffer is successfully allocated, the Driver 301 can use the logical bus address of the common buffer to set up the DMA device.

Input-Output Access Routines 311

The Input-Output Access Routines 311 are defined on and adapted from the common operating system, e.g., the Windows NT and Windows CE programs directly. Again, further details are provided in microfiched Design Guide Appendix C as well as being available commercially from the operating system supplier, e.g., see the Microsoft Windows CE InfoViewer Documentation.

The foregoing description of the preferred embodiment of the present invention has been presented for purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise form or to exemplary embodiments disclosed. Obviously, many modifications and variations will be apparent to practitioners skilled in this art. Similarly, any process steps described might be interchangeable with other steps in order to achieve the same result. The embodiment was chosen and described in order to best explain the principles of the invention and its best mode practical application, thereby to enable others skilled in the art to understand the invention for various embodiments and with various modifications as are suited to the particular use or implementation contemplated.

It is intended that the scope of the invention be defined by the claims appended hereto and their equivalents. Reference to an element in the singular is not intended to mean "one and only one" unless explicitly so stated, but rather means "one or more." Moreover, no element, component, nor method step in the present disclosure is intended to be dedicated to the public

regardless of whether the element, component, or method step is explicitly recited in the following claims.

No claim element herein is to be construed under the provisions of 35 U.S.C. Sec. 112, sixth paragraph, unless the element is expressly recited using the phrase "means for. . .."

5

CLAIMS

What is claimed is:

1. A method for facilitating communication between computer platforms, each of said computer platforms using a like predetermined operating system, and computer peripheral devices, each of said devices having a respective individual device driver system, via a platform layer interface system, said platform layer interface system including a device driver adaptation subsystem associated with said predetermined operating system, the method comprising the steps of:

providing a device driver adaptation subsystem having a library of computer hardware functional abstractions wherein said functional abstractions are substantially common to a plurality of differing types of said computer platforms;

providing a modified platform layer interface system having said device driver adaptation subsystem interfaced with said device driver adaptation subsystem; and

interfacing communications between respective said hardware platforms and said computer peripheral devices, respectively, via said modified platform layer interface system using said device driver adaptation subsystem such that each said device driver system written to said device driver adaptation subsystem is transportable across a plurality of differing types of said computer platforms.

2. The method as set forth in claim 1, the step of providing a device driver adaptation subsystem having a library of computer hardware functional abstractions comprises the step of:

providing a functional abstraction for allocating and for mapping memory substantially concurrently to a call for memory by the device driver system.

3. The method as set forth in claim 2, said step of providing a functional abstraction for allocating and for mapping memory substantially concurrently to a call for memory by the device driver system comprises the steps of:

obtaining and allocating a range of physically contiguous, cache-aligned memory of a device driver system requested size from a non-paged memory pool of the computer platform connected to said modified platform layer interface system, and

sending a mapped, virtual, address pointer to said memory to said device driver system

indicative of a base virtual address for said memory.

4. The method as set forth in claim 2, said step of providing an abstraction for allocating and for mapping memory substantially concurrently to a call for memory by the device driver system, comprises the step of:

providing functionality for the use of memory descriptor lists by said device driver system.

5. The method as set forth in claim 1, the step of providing a device driver adaptation subsystem having a library of computer hardware functional abstractions comprises the step of:

providing a functional abstraction for handling system interrupt allocation.

6. The method as set forth in claim 5, the step of providing a functional abstraction for handling system interrupt allocation comprises the steps of:

permitting access to an arbitrary number of interrupts, and

permitting a device driver system to allocate and connect to an free interrupt vector in the operating system.

7. The method as set forth in claim 5, the step of providing a functional abstraction for handling system interrupt allocation further comprising the step of:

providing system interrupt timing abstractions including an interrupt timer object and a set of subroutines to manipulate said timer object.

8. The method as set forth in claim 5, the step of providing a functional abstraction for handling system interrupt allocation comprising the steps of:

when said device driver system specifies predetermined bus interrupts said device drive system requires notification of, said device driver adaptation subsystem compiling a table associating system interrupts with said bus interrupts, said device driver adaptation subsystem tracking system interrupts, and said device driver adaptation subsystem reporting to said device driver system when a system interrupt occurs and is associated with one of said device driver system specified predetermined bus interrupts.

9. The method as set forth in claim 1, the step of providing a device driver adaptation subsystem having a library of computer hardware functional abstractions comprises the step of:

abstracting platform direct memory access requirements for adaptation by the device driver system.

10. The method as set forth in claim 9, the step of abstracting platform direct memory access requirements for adaptation by the device driver system, comprising the steps of:

allocating direct memory access channels, and

controlling direct memory access operations.

11. The method as set forth in claim 10, the steps of allocating direct memory access channels and controlling direct memory access operations comprising the steps of:

providing adapter object data structures describing computer platforms and peripheral devices,

providing map registers for mapping from a bus-related address to a system-accessible physical memory address, and

providing at least one dedicated adapter channel for buffering data.

12. The method as set forth in claim 11, the step of abstracting platform direct memory access requirements for adaptation by the device driver system comprising the steps of:

when said device driver system calls for a packet-based slave mode direct memory access operation,

setting said dedicated adapter channel,

determining a maximum direct memory access transfer size per direct memory access transfer operation,

determining availability of said adapter channel,

when said adapter channel is available, having said device driver adaptation subsystem conduct said direct memory access transfer operation.

13. The method as set forth in claim 11, the step of abstracting platform direct memory access requirements for adaptation by the device driver system comprising the steps of:

when said device driver system calls for a common buffer slave mode direct memory access operation,

setting a predetermined said adapter channel,

determining a maximum direct memory access transfer size per direct memory access transfer operation,

allocating an accessible buffer for said adapter channel,

when said channel is available, having said device driver adaptation subsystem conduct said direct memory access transfer operation.

14. The method as set forth in claim 11, the step of abstracting platform direct memory access requirements for adaptation by the device driver system comprising the steps of:

when said device driver system calls for a busmaster mode direct memory access operation,

determining a maximum direct memory access transfer size per direct memory access transfer operation,

selecting a driver bus or a system bus for said direct memory access transfer operation,

mapping a bus address to system memory address,

having said device driver adaptation subsystem conduct said direct memory access transfer operation.

15. The method as set forth in claim 1, the step of providing a device driver adaptation subsystem having a library of computer hardware functional abstractions, comprises the step of:

providing input-output access routines between said device driver adaptation subsystem and said computer platforms.

16. The method as set forth in claim 1, wherein said predetermined operating system is a windows-based operating system.

17. A computerized apparatus for conducting interfaced operations between a computer platform, having a predetermined computer operating system wherein said operating system has a computerized platform layer interface system including adaptation means for conforming computer peripheral devices to operate in conjunction with said computer platform, and computer peripheral devices, each having a device driver system, said computerized apparatus comprising:

means for providing a device driver adaptation subsystem having a library of computer hardware functional abstractions wherein said functional abstractions are substantially common to a plurality of differing types of each said computer platform;

means for providing a modified platform layer interface system having said device driver adaptation subsystem therein interfaced with said device driver adaptation subsystem; and

means for interfacing communications between the computer platform and said computer peripheral devices connected thereto, respectively, via said modified platform layer interface system using said device driver adaptation subsystem such that each said device driver system written to said device driver adaptation subsystem is transportable across said plurality of differing types of said computer platforms.

18. The apparatus as set forth in claim 17, the means for providing a device driver adaptation subsystem comprising:

means for providing a functional abstraction for allocating and for mapping memory substantially concurrently to a call for memory by the device driver system.

19. The apparatus as set forth in claim 18, said means for providing a functional abstraction for allocating and for mapping memory substantially concurrently to a call for memory by the device driver system comprising:

means for obtaining and allocating a range of physically contiguous, cache-aligned memory of a device driver system requested size from a non-paged memory pool of the computer platform connected to said modified platform layer interface system, and

means for sending a mapped, virtual, address pointer to said memory to said device driver system indicative of a base virtual address for said memory.

20. The apparatus as set forth in claim 18, said means for providing an abstraction for allocating and for mapping memory substantially concurrently to a call for memory by the device driver system, comprising:

means for providing functionality for the use of memory descriptor lists by said device driver system.

21. The apparatus as set forth in claim 17, the means for providing a device driver adaptation subsystem having a library of computer hardware functional abstractions comprising:

means for providing a functional abstraction for handling system interrupt allocation.

22. The apparatus as set forth in claim 21, the means for providing a functional abstraction for handling system interrupt allocation comprising:

means for permitting access to an arbitrary number of interrupts, and

means for permitting a device driver system to allocate and connect to an free interrupt vector in the operating system.

23. The apparatus as set forth in claim 21, the means for providing a functional abstraction for handling system interrupt allocation further comprising:

means for providing system interrupt timing abstractions including an interrupt timer object and a set of subroutines to manipulate said timer object.

24. The apparatus as set forth in claim 21, the means for providing a functional abstraction for handling system interrupt allocation comprising:

means for determining when said device driver system specifies predetermined bus interrupts said device drive system requires notification regarding,

device driver adaptation subsystem means for compiling a table associating system interrupts with said bus interrupts,

device driver adaptation subsystem means for tracking system interrupts, and

device driver adaptation subsystem means for reporting to said device driver system when a system interrupt occurs and is associated with one of said device driver system specified predetermined bus interrupts.

25. The apparatus as set forth in claim 17, the means for providing a device driver adaptation subsystem having a library of computer hardware functional abstractions comprising:

means for abstracting platform direct memory access requirements for adaptation by the device driver system.

26. The apparatus as set forth in claim 25, the means for abstracting platform direct memory access requirements for adaptation by the device driver system, comprising:

means for allocating direct memory access channels, and means controlling direct memory access operations.

27. The apparatus as set forth in claim 26, the means for allocating direct memory access channels and means for controlling direct memory access operations comprising:

means for providing adapter object data structures describing computer platforms and peripheral devices,

means for providing map registers for mapping from a bus-related address to a system-accessible physical memory address, and

means for providing at least one dedicated adapter channel for buffering data.

28. The apparatus as set forth in claim 27, the means for abstracting platform direct memory access requirements for adaptation by the device driver system comprising:

means for determining when said device driver system calls for a packet-based slave mode direct memory access operation,

means for setting said dedicated adapter channel,

means for determining a maximum direct memory access transfer size per direct memory access transfer operation,

means for determining availability of said adapter channel,

means for signaling to said device driver system when said adapter channel is available,

means for having said device driver adaptation subsystem conduct said direct memory access transfer operation.

29. The apparatus as set forth in claim 27, the means for abstracting platform direct memory access requirements for adaptation by the device driver system comprising:

means for determining when said device driver system calls for a common buffer slave mode direct memory access operation,

means for setting a predetermined said adapter channel,

means for determining a maximum direct memory access transfer size per direct memory access transfer operation,

means for allocating an accessible buffer for said adapter channel,

means for determining when said channel is available,

means for having said device driver adaptation subsystem conduct said direct memory access transfer operation.

30. The apparatus as set forth in claim 27, the means for abstracting platform direct memory access requirements for adaptation by the device driver system comprising:

means for determining when said device driver system calls for a busmaster mode direct memory access operation,

means for determining a maximum direct memory access transfer size per direct memory access transfer operation,

means for selecting a driver bus or a system bus for said direct memory access transfer operation,

means for mapping a bus address to system memory address,

means for having said device driver adaptation subsystem conduct said direct memory access transfer operation.

31. The apparatus as set forth in claim 17, the means for providing a device driver adaptation subsystem having a library of computer hardware functional abstractions, comprising:

providing input-output access routines between said device driver adaptation subsystem and said computer platforms.

32. The apparatus as set forth in claim 17, wherein said predetermined operating system is a windows-based operating system.

33. A computer-to-peripheral platform layer construct for interfacing peripheral device drivers to a variety of types of computer hardware platforms, each of said platforms having a common operating system, comprising:

associated with said operating system, predetermined adaptation layer means for adapting a peripheral device to a predetermined one of said computer platforms; and

interface means for providing an interface between said drivers and each of said types of computer hardware platforms, said interface means, including

memory management means for allocating and for mapping memory substantially concurrently to a call for memory by each of the drivers,

interrupt means for handling system interrupt allocation and for connecting each of the driver to any free interrupt vector in the operating system substantially concurrently to a system interrupt for each of the drivers,

direct memory access means for abstracting platform direct memory access requirements for adaptation by each of the drivers, and

input-output means for interfacing the platform layer construct and each of the platforms,

wherein said construct provides a transparent interface having localized computer hardware functionality such that any of said device drivers programmed to said construct is transportable to a plurality of hardware platforms.

34. The platform layer construct as set forth in claim 33, said memory management means comprising:

means for obtaining and allocating a range of physically contiguous, cache-aligned memory of a device driver system requested size from a non-paged memory pool of the computer platform connected to said modified platform layer interface system, and

means for sending a mapped, virtual, address pointer to said memory to said device driver system indicative of a base virtual address for said memory.

35. The platform layer construct as set forth in claim 33, said interrupt means comprising:

means for specifying an interrupt in terms of a bus-related interrupt vector rather than in

terms of a system interrupt vector.

36. The platform layer construct as set forth in claim 33, said direct memory access means comprising:

for slave mode direct memory access modes, means for maintaining state control of a platform direct memory access controller.

37. The platform layer construct as set forth in claim 33, said direct memory access means comprising:

for a busmaster direct memory access modes, means for mapping from busmaster bus addresses to platform memory addresses.

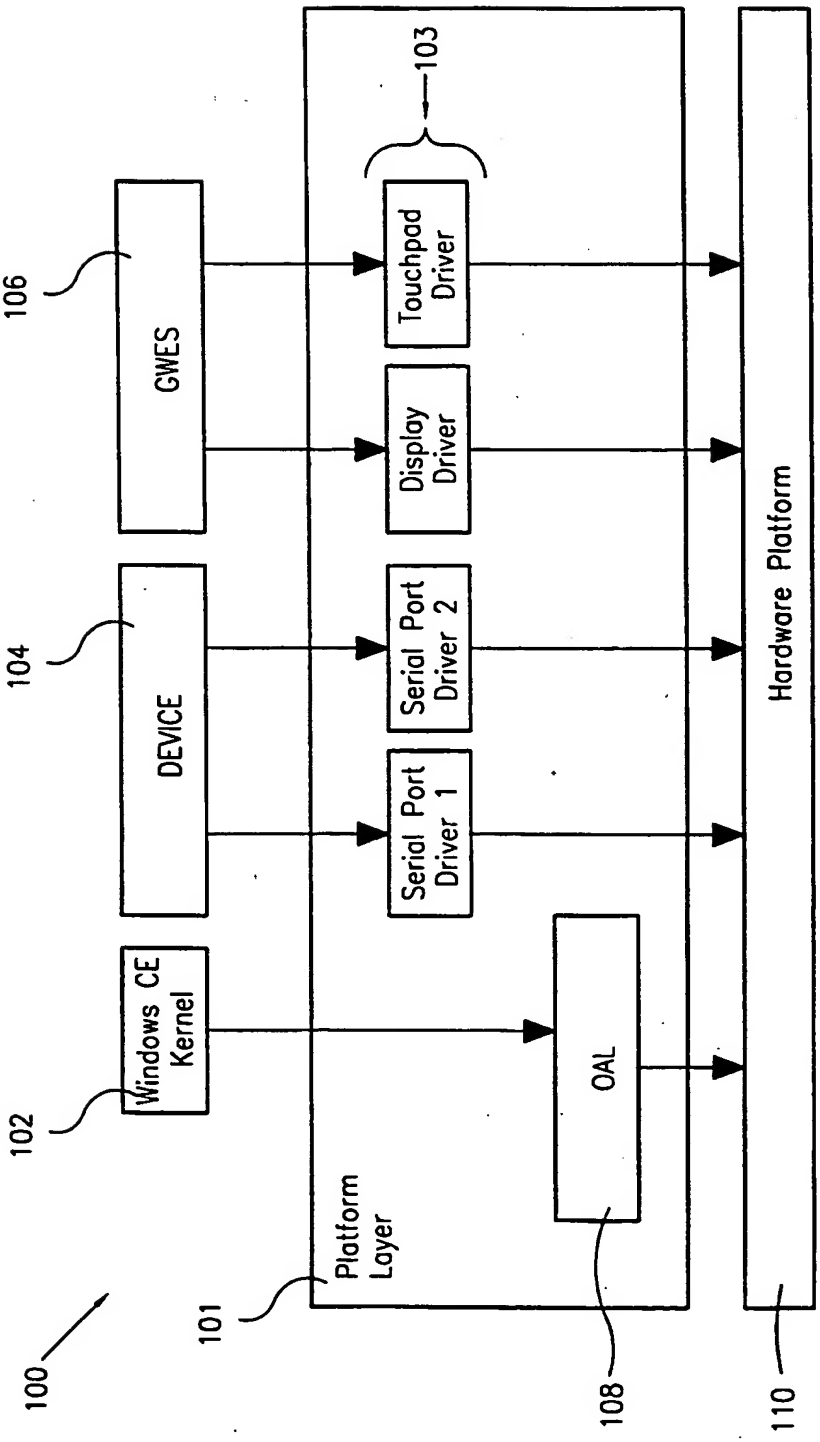


FIGURE 1
(Prior Art)

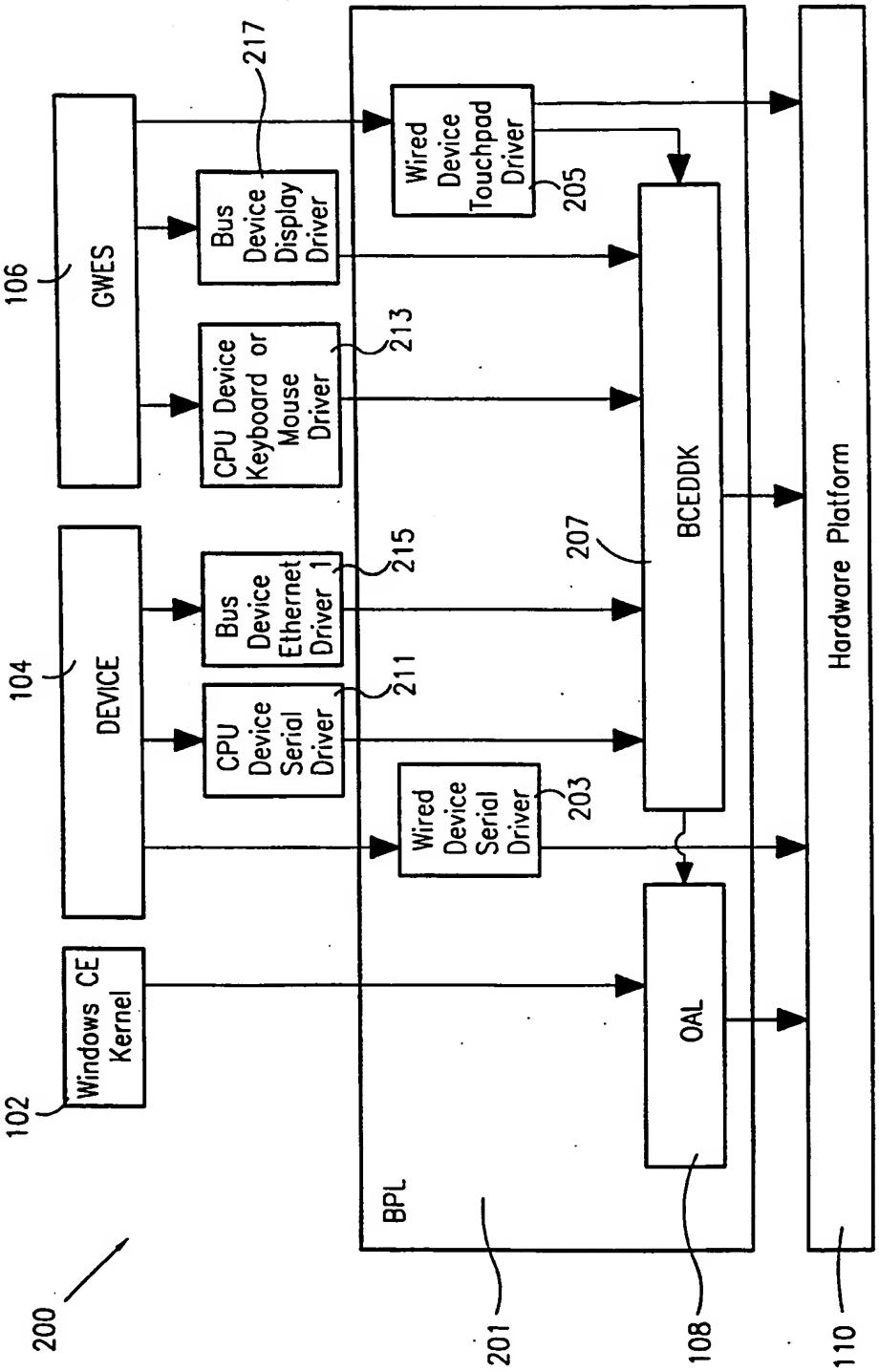

200
102
Windows CE Kernel
104
DEVICE
106
GWES
BPL
201
108
OAL
110
Hardware Platform
BCEDDK
207
Wired Device Touchpad Driver
205
Bus Device Display Driver
217
CPU Device Keyboard or Mouse Driver
213
Bus Device Ethernet Driver 1
215
CPU Device Serial Driver
211
Wired Device Serial Driver
203

FIGURE 2

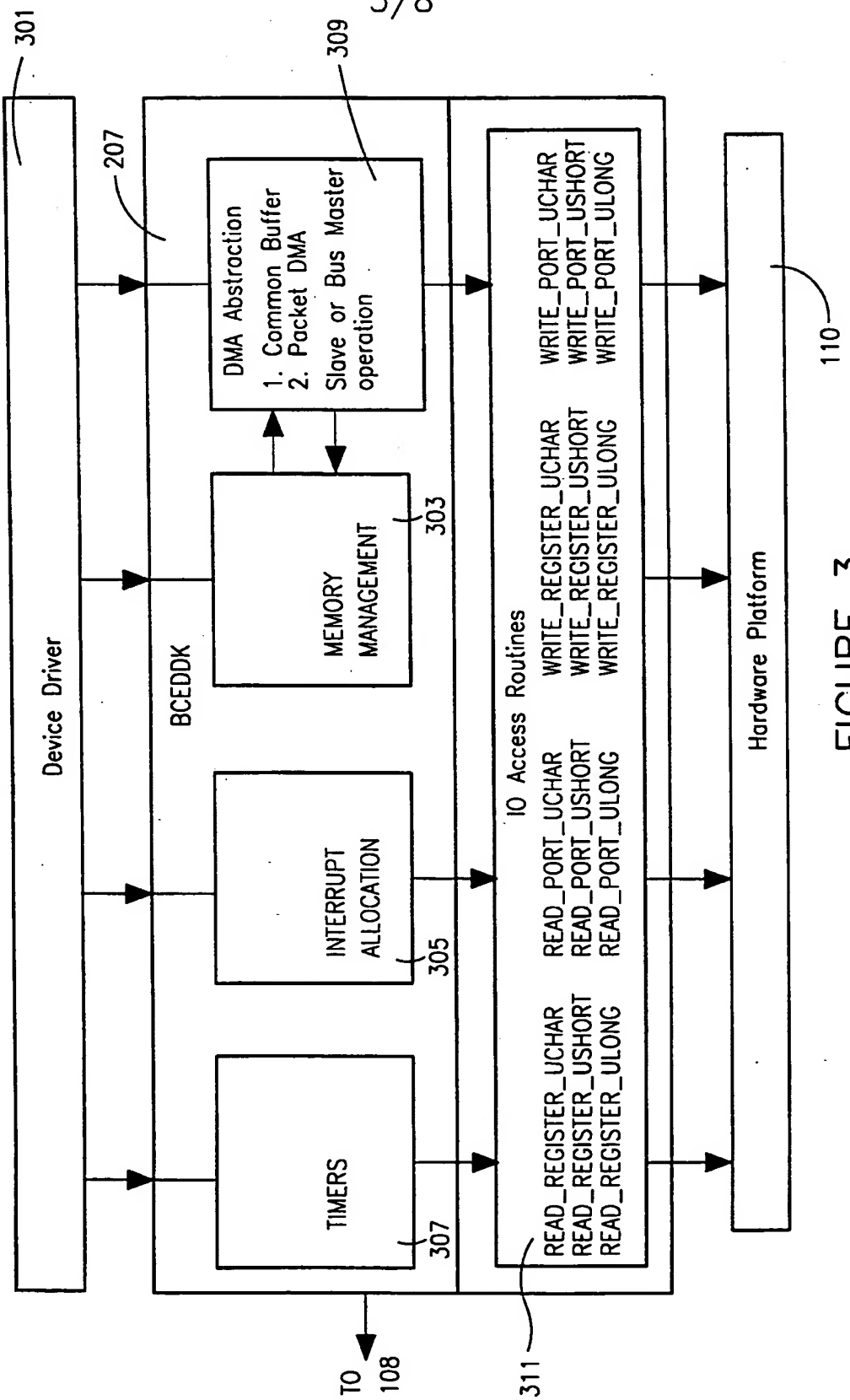

301
207
309
DMA Abstraction
1. Common Buffer
2. Packet DMA
Slave or Bus Master
operation
Device Driver
BCEDDK
MEMORY
MANAGEMENT
303
INTERRUPT
ALLOCATION
305
TIMERS
307
IO Access Routines
READ_PORT_UCHAR
READ_PORT_USHORT
READ_PORT_ULONG
WRITE_PORT_UCHAR
WRITE_PORT_USHORT
WRITE_PORT_ULONG
READ_REGISTER_UCHAR
READ_REGISTER_USHORT
READ_REGISTER_ULONG
WRITE_REGISTER_UCHAR
WRITE_REGISTER_USHORT
WRITE_REGISTER_ULONG
Hardware Platform
110
TO
108
311

FIGURE 3

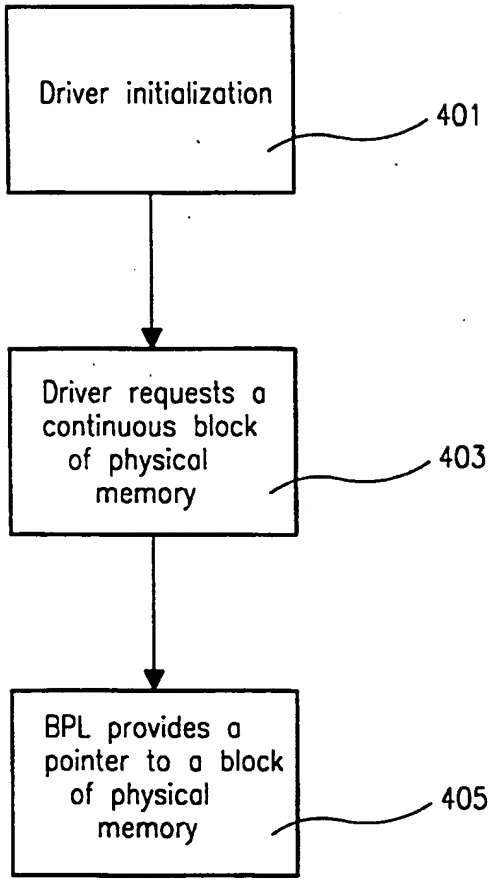

Driver initialization
401
Driver requests a continuous block of physical memory
403
BPL provides a pointer to a block of physical memory
405

FIGURE 4

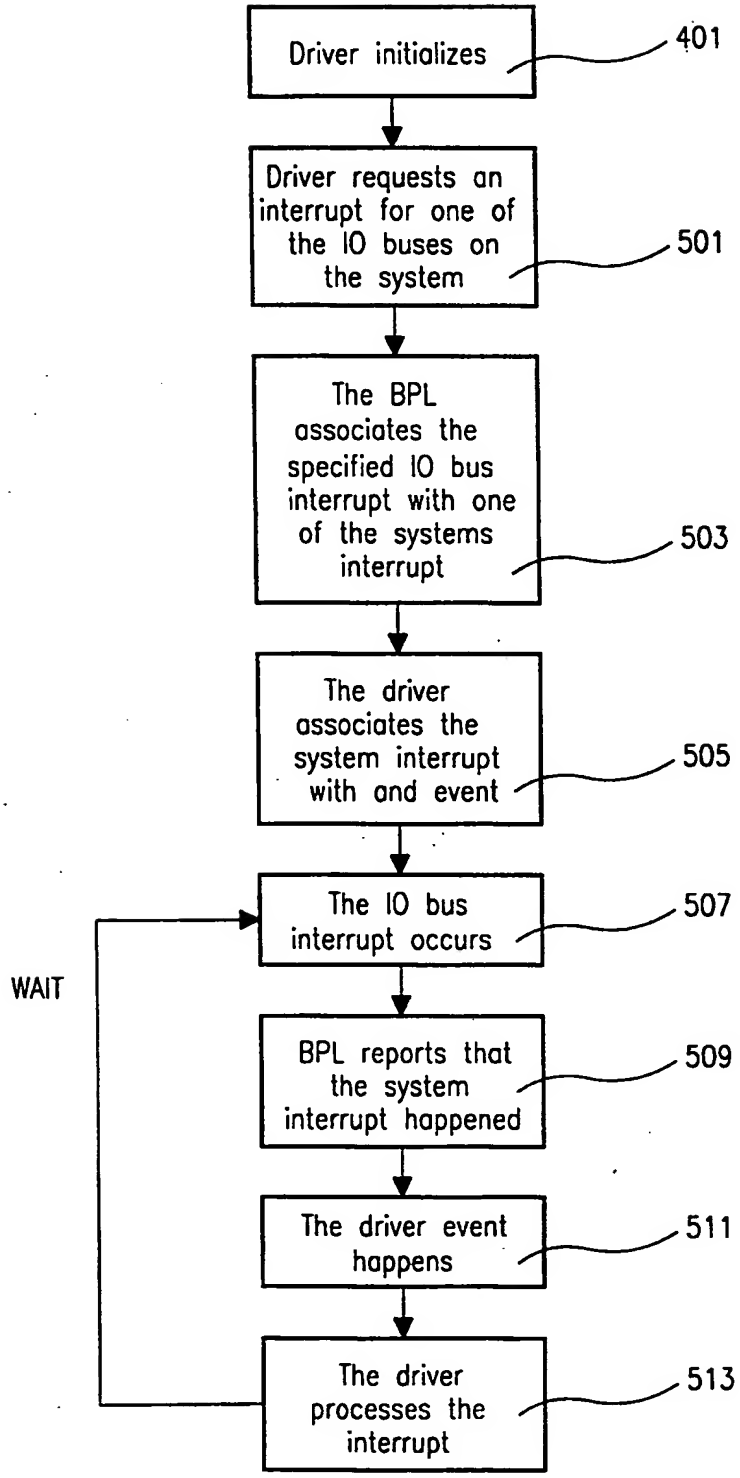

Driver initializes
401
Driver requests an interrupt for one of the IO buses on the system
501
The BPL associates the specified IO bus interrupt with one of the systems interrupt
503
The driver associates the system interrupt with and event
505
The IO bus interrupt occurs
507
WAIT
BPL reports that the system interrupt happened
509
The driver event happens
511
The driver processes the interrupt
513

FIGURE 5

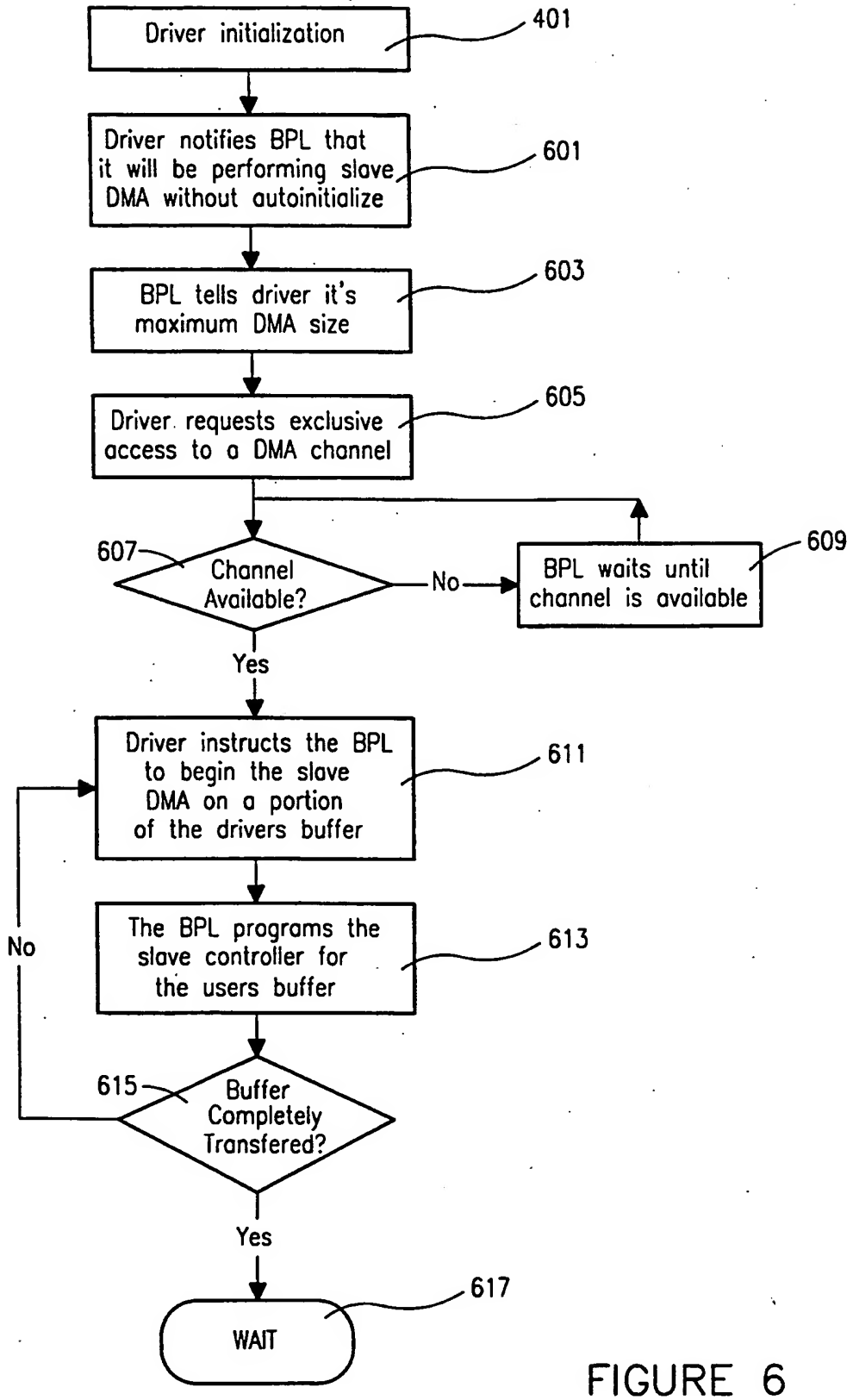

Driver initialization
401
Driver notifies BPL that it will be performing slave DMA without autoinitialize
601
BPL tells driver it's maximum DMA size
603
Driver requests exclusive access to a DMA channel
605
607
Channel Available?
No
BPL waits until channel is available
609
Yes
Driver instructs the BPL to begin the slave DMA on a portion of the drivers buffer
611
The BPL programs the slave controller for the users buffer
613
No
615
Buffer Completely Transfered?
Yes
617
WAIT

FIGURE 6

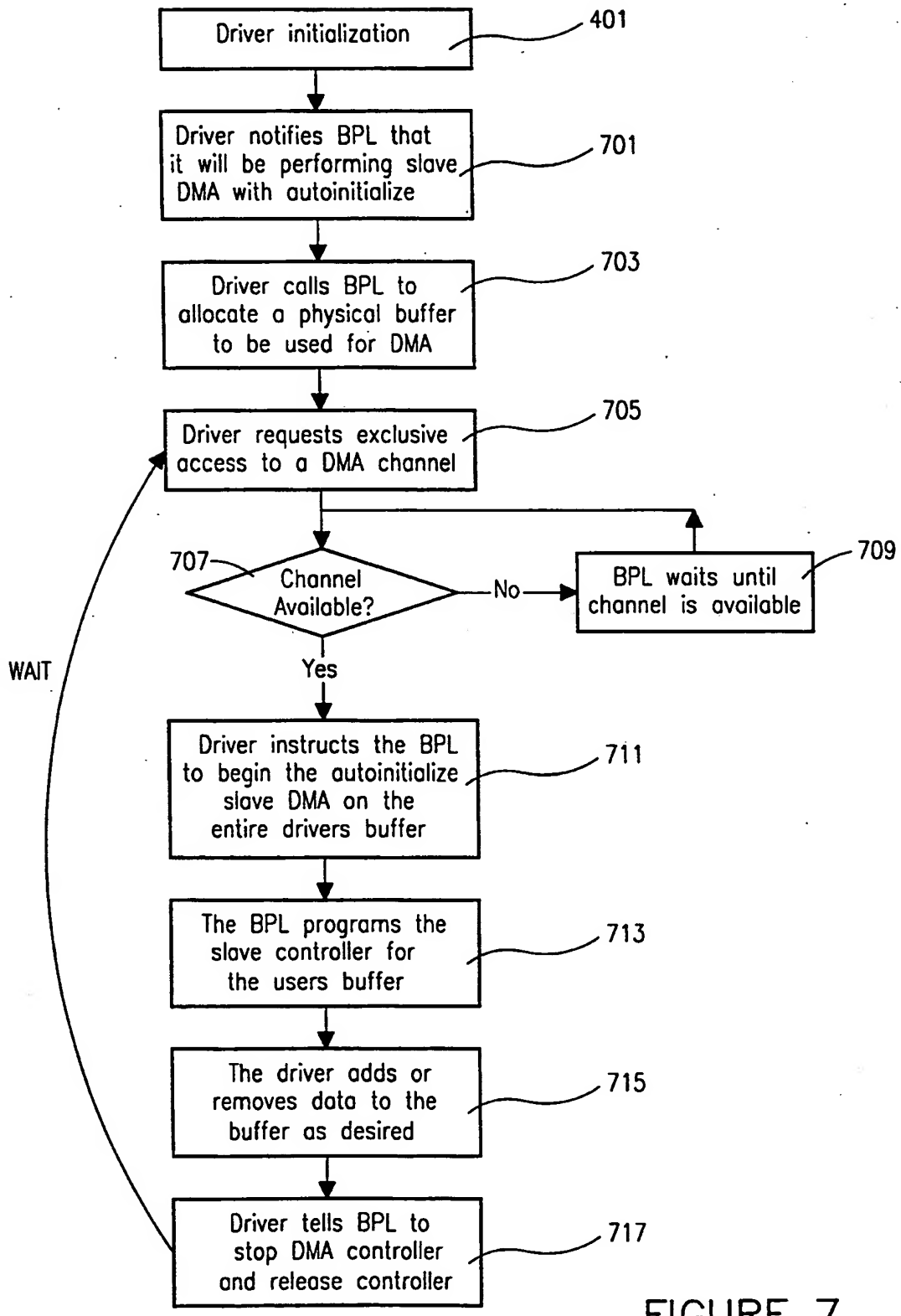

Driver initialization
401
Driver notifies BPL that it will be performing slave DMA with autoinitialize
701
Driver calls BPL to allocate a physical buffer to be used for DMA
703
Driver requests exclusive access to a DMA channel
705
707
Channel Available?
No
BPL waits until channel is available
709
Yes
WAIT
Driver instructs the BPL to begin the autoinitialize slave DMA on the entire drivers buffer
711
The BPL programs the slave controller for the users buffer
713
The driver adds or removes data to the buffer as desired
715
Driver tells BPL to stop DMA controller and release controller
717

FIGURE 7

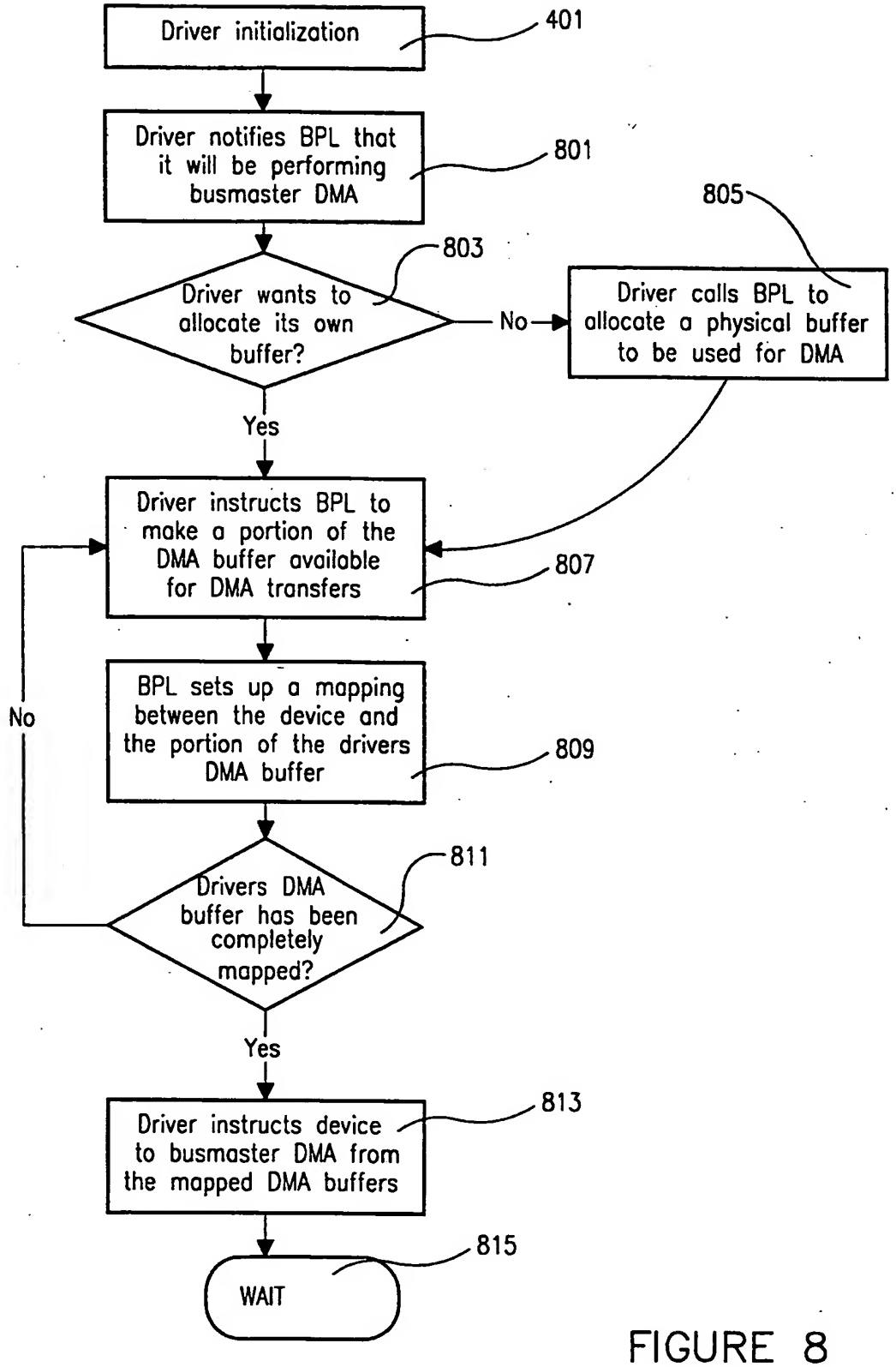

Driver initialization
401
Driver notifies BPL that it will be performing busmaster DMA
801
805
803
Driver wants to allocate its own buffer?
No
Driver calls BPL to allocate a physical buffer to be used for DMA
Yes
Driver instructs BPL to make a portion of the DMA buffer available for DMA transfers
807
BPL sets up a mapping between the device and the portion of the drivers DMA buffer
809
No
811
Drivers DMA buffer has been completely mapped?
Yes
Driver instructs device to busmaster DMA from the mapped DMA buffers
813
815
WAIT

FIGURE 8

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
7 December 2000 (07.12.2000)

PCT

(10) International Publication Number
WO 00/74368 A3

(51) International Patent Classification[7]: G06F 13/10

(21) International Application Number: PCT/US00/15416

(22) International Filing Date: 1 June 2000 (01.06.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/324,073 1 June 1999 (01.06.1999) US

(71) Applicant: BSQUARE CORPORATION [US/US]; 3150 139th Avenue S.E., Suite 500, Bellevue, WA 98005-4081 (US).

(72) Inventors: BROOKS, Steve; 21433 SE 19th Street, Issaquah, WA 98029 (US). MURRAY, Jason; 23803 NE 27th Street, Redmond, WA 98053 (US). RICHARDS, David; 4155 145th Avenue, NE, Bellevue, WA 98007 (US).

(74) Agent: LARIVIERE, GRUBMAN & PAYNE, LLP; P.O. Box 3140, Monterey, CA 93942 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

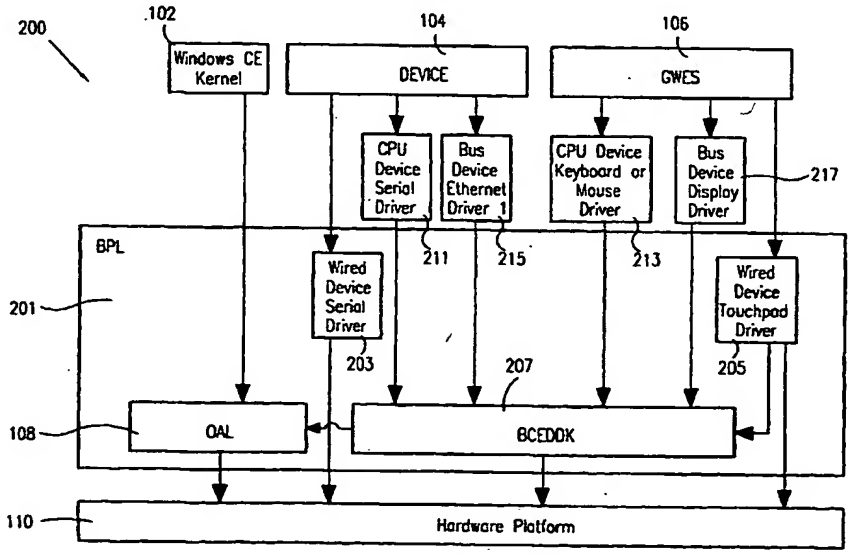
(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

(88) Date of publication of the international search report:
9 August 2001

[Continued on next page]

(54) Title: DEVICE DRIVER PLATFORM LAYER



(57) Abstract: A Platform Layer interface for device drivers - also referred to hereinafter as the bSquaretm Platform Layer, or "BPL", (201) - conforms a computers operating system - e.g., a WINDOWS CE operating system (102) - and provides a transportable system for connecting device drivers to a variety of computer hardware platform (110). The BPL provides an interface that imposes structure on the operating system platform layer with several functionalities as individual components of the BPL used by the drivers, including: 1) a Memory Management component, 2) an Interrupt Allocation component, 3) an Interrupt Timers component, 4) a Direct Memory Access ("DMA") component, including slave and master bus operations, and 5) an Input-Output Access Routines component.

WO 00/74368 A3

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.